Recursive double-size fixed precision arithmetic

Alexis Breust, Christophe Chabot, Jean-Guillaume Dumas, Laurent Fousse, Pascal Giorgi

Recursive double-size fixed precision arithmetic

Alexis Breust[1], Christophe Chabot[1], Jean-Guillaume Dumas[1], Laurent Fousse[1], Pascal Giorgi[2]

[1] Laboratoire J. Kuntzmann, Université Grenoble Alpes. France[‡]
firstname.lastname@imag.fr.
[2] LIRMM, CNRS, Université Montpellier, France[‡]
pascal.giorgi@lirmm.fr.

Abstract. We propose a new fixed precision arithmetic package called RecInt. It uses a recursive double-size data-structure. Contrary to arbitrary precision packages like GMP, that create vectors of words on the heap, RecInt large integers are created on the stack. The space allocated for these integers is a power of two and arithmetic is performed modulo that power. Operations are thus easily implemented recursively by a divide and conquer strategy. Among those, we show that this packages is particularly well adapted to Newton-Raphson like iterations or Montgomery reduction. Recursivity is implemented via doubling algorithms on templated data-types. The idea is to extend machine word functionality to any power of two and to use template partial specialization to adapt the implemented routines to some specific sizes and thresholds. The main target precision is for cryptographic sizes, that is up to several tens of machine words. Preliminary experiments show that good performance can be attained when comparing to the state of art GMP library: it can be several order of magnitude faster when used with very few machine words. This package is now integrated within the Givaro C++ library and has been used for efficient exact linear algebra computations.

1 Introduction

Mathematical computations that needs integers above machine word precision are compelled to rely on a third party library. Among arbitrary precision libraries for integers, GMP [11] (or its fork MPIR - www.mpir.org) is the most renown and efficient. The underlying structure of GMP/MPIR integers is based on an array of machine word integers that are accessed through a pointer. For instance, a 256-bits integers a can been represented by a dimension four array $[a_0, a_1, a_2, a_3]$ such that $a = a_0 + a_1 2^{64} + a_2 2^{128} + a_1 2^{192}$. Additionally to the pointer, GMP/MPIR stores two integers that represent respectively the number of allocated words and the number of used words, ensuring the dynamic of the precision. Indeed, such a structure is well designed to efficiently handle arbitrary precision but can be too costly when one knows in advance the targeted

[‡] This material is based on work supported in part by the Agence Nationale pour la Recherche under Grant ANR-11-BS02-013 HPAC.

precision. Furthermore, when dealing with multiple integers at the same time, i.e. in matrix computation, access through pointers breaks cache mechanisms and penalizes performances. Hopefully, one can access to a low level API (`mpn` level) to both handle fixed precision and multiple integer without suffering from an heavy interface. However, this approach let the memory management to the user and cannot be incorporated to code that have been designed at a higher level. The purpose of our work is to provide an alternative to the GMP library for fixed precision integers that allow the flexibility of a high level library while still being efficient. In particular, our approach is to design very simple codes that extend naturally machine word to other powers of two.

A common and efficient way to compute over prime finite fields is to first perform the operation over the integers and map the result back to the field via modular reduction. In this specific case, the use of arbitrary precision is not relevant since precision is fixed by the cardinality of the finite field. Indeed, doubling the precision is often sufficient. To further optimize this approach, one can use precompiled code to tackle specific range of modulus. This approach has been proposed in the MPFQ library for cryptographic size moduli [10]. While being very efficient to perform scalar computation over finite field, i.e. modular exponentiation, the design of this library does not allow to use lazy modular reduction that proved very efficient for linear algebra [7].

Besides fixed precision integers, our package also provides prime fields support that can be easily embedded in high level code, while offering very good performance, in particular for matrix computations.

2 Functionality

2.1 Fixed precision arithmetic : extending the word size

With fixed precision integers, the maximum number of bit is given in advance and all arithmetic operations are done within this precision. In particular, if K is the maximum bitsize (i.e. the precision) then all computations are done modulo 2^K. This corresponds to forgetting the carry for addition/subtraction or to getting the lowest K-bits part of the integer products.

Fixing K in advance facilitates the storage of integers through static array of size $\lceil K/64 \rceil$ on 64-bits architecture. In order to mimic word-size integers and to fully use the memory, we focus only on supporting integers of size a power of two: $K = 2^k$, with $k >= 6$. This assumption will allow us to provide a simple recursive data structure that eases manipulation and implementation of most arithmetic operations as explained in Section 3.1.

2.2 Fixed precision types

Our C++ package is devoted to provide functionality for fixed integer types. As for native integers, we provide signed and unsigned types. Furthermore, we extend this by providing a modular integer type that allows, e.g., computation over prime finite fields as a standard type. We denote by `ruint<k>`

the type of our unsigned integer at precision 2^k (for instance we also define: `using ruint128 = ruint<7>; using ruint256 = ruint<8>;` etc.). Then the other types (signed and modular) are implemented with this structure. Our fixed precision integers are integrated in the Givaro library and available at https://github.com/linbox-team/givaro.

Signed type. The recursive signed integer is defined as a `rint<k>` type. The data structure is just a `ruint<k>` and operations are performed via unsigned operations thanks to a two's complement representation.

Modular integer type. A special type `rmint<k>` is provided for modular operations. The data structure is composed of an unsigned value, `ruint<k>`, as well as a global `ruint<k>` for the modulus that is shared by all modular elements. Modular elements modulo p are represented in the range $[0, p-1]$ and all operations guarantee that values are always reduced to this range.

2.3 Integer operations

Our package provides basic integer operations defined both as mathematical operator $(\times, +, -, /)$ or with their functional variants (mul,add,sub,div). A simple example of using our package for doing vector dot product with 256-bit integers is given below:

```
ruint<8> *A,*B,res=0; ... for(size_t i=0;i<n;i++) res+=A[i]*B[i];
```

Elements of type `ruint<8>` can be easily converted to `rint<8>` for computing with signed integer, or to `rmint<8>` for computing modulo a 256-bit prime. Several other functions are available (gcd, axpy, exponentiation, comparison, shift) as well as some internal functions that eases algorithm implementation, e.g. addition with carry, internal access to the structure, etc.

Two types of modular reduction. One can easily switch between two type of modular reduction. In particular, one can choose between classic modular reduction using Euclidean division or the one using Montgomery's method [13], described in Section 3.2. The choice of representation is done at compile time, by a template constant: `rmint<k>`, or `rmint<K, MG_INACTIVE>` gives you the classical one, while `rmint<K, MG_ACTIVE>` uses Montgomery's reduction.

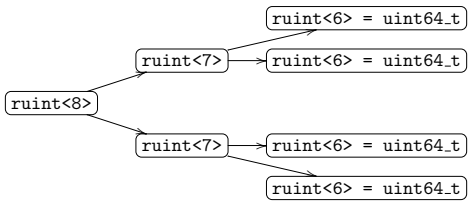
3 Underlying theory and technical contribution

One drawback of arbitrary precision integers is to use a dynamic structure to handle the variation of integers value. There, defining contiguous structure of such integers leads to break cache prefetching when accessing data. Generally, one alternative is then to use a low level API where dynamics of the integers is delegated to the programmer. Unfortunately, this often becomes incompatible with the design of code at a high level, and then may not allow to re-use existing code. In the following section, we present our prototype designing a simple data structure preserving good performances.

3.1 Template recursive data structure

Using specificities of fixed precision integers, we can represent an integer of 2^k bits as two integers of 2^{k-1} bits and therefore use a recursive representation. `ruint<k>` is our recursive unsigned integer of 2^k bits. The base case `ruint<6>` is called a limb and it corresponds to a native 64-bits integer. With these conventions, a `ruint<k>` can store any value between 0 and $2^{2^k} - 1$. In order to ensure a static contiguous storage, we chose to use a template recursive data structure with partial specialization. Note that the compiler will always completely unroll this structure at compiled time.

Our C++ template structure `ruint<>` is given hereafter together with an example for 256-bits integers, where `High` and `Low` correspond respectively to the leading and trailing 2^{k-1}-bits of the 2^k-bit integer.




```
template <size_t k>
struct ruint {
  ruint<k-1> High, Low;
};

template <> struct ruint<6>
{uint64_t Value;};
```

3.2 Arithmetic operations.

Thanks to the recursive structure of `ruint<k>`, the implementation of arithmetic operations can be done by a recursive approach. For instance, addition can be done by two recursive calls with a carry propagation in between. Of course, the base case is mapped to the corresponding word-size integer operation. One major interest of such approach is that the compiler will unroll all the recursive calls leading to reduced control flow overhead and better instruction scheduling. As shown in Figure 1 (left), such approach leads to better performance against the GMP library for the addition of two integers (when the result is stored in one of the operand) up to 1024 bits.

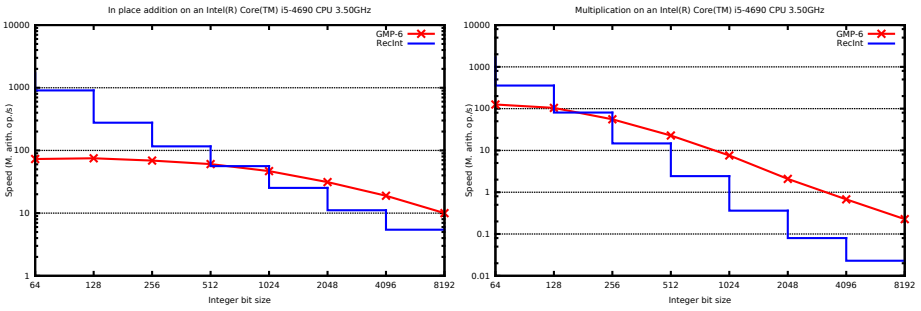


Fig. 1. Comparing integer addition (left) and multiplication (right) with GMP library

Multiplication is handled via a naive approach (or via Karatsuba's method), still using a recursive implementation, as shown in the following code:

```
template <size_t K> inline void lmul_naive
(ruint<K>& ah, ruint<K>& al, const ruint<K>& b, const ruint<K>& c) {
        bool rmid, rlow; ruint<K> bcmid, blcl;
        lmul_naive(blcl, b.Low, c.Low);              // Low part
        lmul_naive(bcmid, b.High, c.Low);            // Middle part
        laddmul(rmid, bcmid, b.Low, c.High, bcmid);  // Middle part
        laddmul(ah, b.High, c.High, bcmid.High);     // High part
        copy(al.Low, blcl.Low);
        add(rlow, al.High, blcl.High, bcmid.Low);
        if (rlow)  add_1(ah);
        if (rmid)  add_1(rmid, ah.High);
    }
```

Still, without using any specific assembly code, contrary to GMP, our approach allows to have good speed-up for small integers up to 256-bits, as shown in Figure 1 (right). Remark: We use Karatsuba's method only for more than 1024 bits.

Montgomery's modular reduction. Classical modular reduction uses integer division to map the result back into the desired integer range. Besides good theoretical complexity [2], this approach is not efficient in practice since hardware divisions are costly. If there exists a radix β such that divisions with β are inexpensive and $gcd(\beta, M) = 1$, Montgomery gives in [13] a method for modular reduction without trial division. The idea of Montgomery is to compute $C\beta^{-1} \bmod M$ instead of computing $C \bmod M$. As shown in [2, Algorithm 2.7], the fast version of this reduction requires to compute integer products modulo β and integer division by β. Assuming β to be a power of 2, this provides an algorithm without any division.

When the integer C to be reduced has a precision 2^{k+1} and the modulo has a precision of 2^k, i.e. $\beta = 2^{2^k}$, this boils down to two multiplications at precision 2^k. The design of our recursive integers is naturally compliant to such method and implementation is almost straightforward. From our first experiment, this straightforward implementation gives 1.5 speedup against GMP for 128-bits integer but becomes not competitive for larger modulus.

Inversion modulo 2^{2^k}. On of the main operation in the setup of Montgomery reduction is the computation of the inverse of M modulo $\beta = 2^{2^k}$. The classical algorithm for this is to use the extended Euclidean algorithm. But with the special structure of B, it is better to use a Newton-Raphson iteration [1,5], doubling the precision at each iteration. Thus here also our recursive structure is perfectly suited to this kind of algorithm, as shown in Figure 2 (left). Note that GMP uses an extended gcd.

Recursive division. For the integer division, our structure is also well suited to a recursive algorithm. Thus, we use that of [3] which uses two sub-algorithms dividing respectively 2 digits by 1 digit and 3 halves by 2. They allow then a recursive division of an s-digits integer by a r-digits integer with complexity $O(rs^{\log(3)-1} + r\log(s))$.

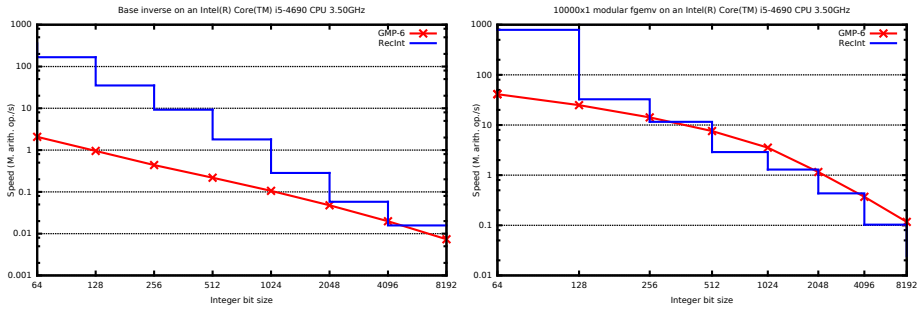


Fig. 2. Newton-Raphson iteration for the inverse modulo a prime power (left) and Modular matrix-vector multiplication (right)

Recursive shift. One operation is slightly more complicated on a recursive structure: shifting. At each recursive level, five cases have to be explored: no shift; shifting the lower part on the high part, in parts or completely; shifting more than half the word; shifting more than the whole word. For instance, if the shift d is such that $0 < d < 2^{2^{k-1}}$, then $b = a << d$ satisfies: $b.High = (a.High \ll d) \oplus \left(a.Low \gg (2^{k-1} - d)\right)$ and $b.Low = (a.Low \ll d)$.

4 Application

4.1 Dense linear algebra: Freivalds certificate

For dense linear algebra, a basic building block is the matrix-vector multiplication. We show in Figure 2 (right) a multi-precision version of a modular matrix-vector multi-precision. There, we use the FFLAS-FFPACK package [7], version 2.2.1 (`http://linalg.org/projects/fflas-ffpack`) and just change the underlying representation from GMP to RecInt.

This possibility of easily changing the base representation can also be extremely important when certifying the results. In verified computing, a client (the Verifier) will check the results provided by a server (typically a cloud, the Prover). In dense linear algebra the basic tools for this is Freivalds' certificate for matrix-multiplication [9]. It uses matrix-vector multiplications with random vectors to check that $C = AB$, via $Cv - A(Bv) \stackrel{?}{=} 0$ on a random projection. With this certificate it is then possible to check any fast dense linear algebra computations with any precision [12, § 5]. These certificates have a double goal. First this is a way to improve the confidence in a result, and this is even more the case if the underlying data structure is different for the computation and for the certificate. Second it provides a way to check outsourced computations. This is economically viable only if the Verifier's time is faster than the Prover's time. We show in Figure 2 (right) that RecInt makes it possible to gain on all dense linear algebra verification, when operations require a few machine words.

4.2 Sparse Linear algebra: prime power rank

The Smith normal form is useful, in topology for instance, for computing the homology of a simplicial complex over the integers. There, the involved matrices are quite often sparse and computations reduce to computing the rank of these matrices modulo some prime powers [8, § 5.1]. We report in Table 1 on timings using the LinBox library [6], version 1.4.1 (`https://github.com/linbox-team/linbox`). The last invariant factor of the matrix S16.231x231 is 2^{63}, while that of S22.1002x1002 is 2^{85}, so both need some extra precision to be computed. But as this factor is not known in advance, a strategy is to double the precision, until the rank modulo the prime power reaches the integer rank of the matrix. For the considered matrices we would stop at 256 bits, but we see in Table 1 that the strategy can be faster until 512 bits. As a comparison we also give timings modulo 3^{40} to show that this is not specific to the characteristic 2.

Table 1. Rank modulo a prime power `http://hpac.imag.fr/Matrices/Tsuchioka`

Matrix	mod	int64_t	RecInt					GMP
			6	7	8	9	10	
S16.231x231	3^{40}	-	**0.01s**	0.05s	0.12s	0.37s	1.28s	0.19s
S16.231x231	2^{63}	**0.03s**	**0.03s**	0.09s	0.29s	1.15s	4.64s	1.85s
S16.231x231	2^{64}	-	-	**0.09s**	0.30s	1.15s	4.64s	1.85s
S22.1002x1002	3^{40}	-	**0.34s**	1.04s	2.60s	7.67s	25.84s	6.32s
S22.1002x1002	2^{63}	-	**2.60s**	7.00s	23.10s	88.24s	356.79s	154.22s
S22.1002x1002	2^{86}	-	-	**7.37s**	24.17s	90.30s	357.44s	190.36s

4.3 Towards an FPGA implementation

We present here the first attempts towards an implementation on a FPGA. To build a programmable hardware (`http://shiva.minalogic.net/`), we had to provide basic arithmetic libraries to be used in an Elliptic curve based encryption scheme. We chose to use a dedicated software transforming C++ source into VHDL called GAUT [4]. The creation of a VHDL program can be split in the following steps: compilation of the C++ source and creation of the corresponding graph; compilation of the library containing the required operations; synthesis of the VHDL program and estimation of performance.

Table 2. FPGA area of modular exponentiation for different output flows, with RecInt integers from 128 to 512 bits

128 bits	op/s	7812	15625	31250	62500	125000	250000
	Flipflops	3040	6100	7008	9538	16617	32199
256 bits	op/s	976	1953	3906	7812	15625	31250
	Flipflops	3618	4391	6923	7654	8776	14542
512 bits	op/s	61	122	244	488	976	1953
	Flipflops	3553	3553	3553	4704	5729	7458

Table 2 shows some simulations of a modular exponentiation on a Virtex 5. We made the output flow vary in order to check the effect on the required size on the FPGA. We notice that required size can be significantly reduced if we accept a lower output flow. One nice point is that, due to the simple recursive structure of RecInt, these results have been obtained without significant modifications on the C++ source and automatically transformed. They are not optimal but rather promising, with no significant work load.

5 Conclusion and Perspective

Our RecInt package is a first attempt to provide high level API for fixed precision integers that are usable out of the box. Thus it is easy to switch from native integer types and to still provide good performance when compared to standard libraries. To further improve the performance, we need to introduce SIMD vectorized instructions in the design of our specializations and remove as much as possible conditional jumps. Modular multiplication should also benefit from either Barret's method or Montgomery-Svoboda's algorithm [2, §2.4.2].

References

1. O. Arazi and H. Qi. On calculating multiplicative inverses modulo 2^m. *IEEE Transactions on Computers*, 57(10):1435–1438, Oct. 2008.
2. R. Brent and P. Zimmermann. *Modern Computer Arithmetic*. Cambridge University Press, New York, NY, USA, 2010.
3. C. Burnikel and J. Ziegler. Fast recursive division. Technical Report MPI-I-98-1-022, Max Planck Institute fr Informatik, Oct. 1998.
4. P. Coussy, C. Chavet, P. Bomel, D. Heller, E. Senn, and E. Martin. *High-Level Synthesis: From Algorithm to Digital Circuit*, chapter GAUT: A High-Level Synthesis Tool for DSP Applications, pages 147–169. Springer Netherlands, 2008.
5. J.-G. Dumas. On Newton-Raphson iteration for multiplicative inverses modulo prime powers. *IEEE Transactions on Computers*, 63(8):2106–2109, Aug. 2014.
6. J.-G. Dumas, T. Gautier, M. Giesbrecht, P. Giorgi, B. Hovinen, E. Kaltofen, B. D. Saunders, W. J. Turner, and G. Villard. LinBox: A generic library for exact linear algebra. In *ICMS'2002, Beijing, China*, pages 40–50, Aug. 2002.
7. J.-G. Dumas, P. Giorgi, and C. Pernet. Dense linear algebra over prime fields. *ACM Transactions on Mathematical Software*, 35(3):1–42, Nov. 2008.
8. J.-G. Dumas, B. D. Saunders, and G. Villard. On efficient sparse integer matrix Smith normal form computations. *J. Symb. Comp.*, 32(1/2):71–99, July–Aug. 2001.
9. R. Freivalds. Fast probabilistic algorithms. In J. Bečvář, editor, *Math. Foundations of Comp. Sc.*, pages 57–69, Olomouc, Czechoslovakia, 1979. Springer-Verlag.
10. P. Gaudry and E. Thomé. The mpFq library and implementing curve-based key exchanges. In *SPEED: Software Performance Enhancement for Encryption and Decryption*, pages 49–64, Amsterdam, Netherlands, June 2007. ECRYPT Network.
11. T. Granlund. *The GNU multiple precision arithmetic library*, Nov. 2015. v6.1.
12. E. L. Kaltofen, M. Nehring, and B. D. Saunders. Quadratic-time certificates in linear algebra. In *ISSAC'2011, San Jose, USA*, pages 171–176, June 2011.
13. P. L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, Apr. 1985.